Wprowadzenie do narzędzia JUnit

Mateusz Miotk

19/20 luty 2018

Instytut Informatyki UG

Geneza testowania

Z każdym projektem informatycznym wiąże się wiele różnych działań określonych mianem testowania, w których niektóre wymagają udziału przyszłych użytkowników, programów, a inne pracy specjalnego zespołu.

Tutaj będziemy skupiać się ogólnie na **testach jednostkowych**, ponieważ są prostym i tanim sposobem pozwalającym tworzyć niezawodny kod.

Kiedy testować

Wiele orgranizacji kładzie nacisk na testowanie oprogramowanie w **końcowej** fazie projektu, gdyż presja terminów w praktyce zmniejsza szanse właściwego przetestowania kodu.

Wielu programistów uważa, że testowanie to niepotrzebny obowiązek - oni są tylko od pisania kodu. To błędne rozumowanie, gdyż programista najlepiej potrafi przetestować swój kod.

Ogólnie im więcej testów tym lepiej, aczkolwiek nie należy powielać tych samych testów w mnogich ilościach.

Kiedy testować - wprowadzenie do TDD

Tutaj będzie pokazywana technika TDD - **Test-Driven Development**, która polega na tym, że **najpierw** piszemy test (najczęściej jednostkowy), a następnie piszemy kod, aby ten test przeszedł.

TDD ma dużo zalet do których należą między innymi:

- Skraca czas wprowadzanych produktów
- Ułatwia refaktoryzacje (czyli optymalizowanie kodu)
- Sprzyja luźnemu łączeniu komponentów (fragmentów kodu)
- Dokumentacja kodu
- Brak konieczności debugowania
- Symulacje działań (poprzez stosowanie atrap - EasyMock, Mockito itd.)

Ogólny cykl w TDD

TDD to proces opraty na powtarzaniu bardzo krótkiego cyklu prac programistycznych. Wykorzystywane jest tu podejście z **XP** (eXtreme programming), zachęcająca do tworzenia prostych projektów, które z wysokim prawdopodobieństwem są prawidłowe.

Myśląc o TDD należy mieć na myśli następujace etapy (wyjaśnione później):- **Czerwone, Zielone, Refaktoryzacja**.

Natomiast procedura napisania testu jest następująca:

1. Napisz test.
2. **Uruchom** wszystkie testy.
3. Napisz kod rozwiązania.
4. Uruchom wszystkie testy.
5. Przeprowadź refaktoryzację.
6. Uruchom wszystkie testy.

Czerwone, zielone, refaktoryzacja

W trakcie pisania testów rozwiązanie znajduje się w stanie **czerwone**, ponieważ test kończy się niepowodzeniem.

Po zakończeniu kodu rozwiązania powiązanego z testem wszystkie testy powinny zakończyć się powowdzeniem (stan **zielone**).

Następnie powinno się przeprowadzić refaktoryzację kodu (stan refaktoryzacji).

Porady przy podejściu TDD

Nie traktuj kodu napisanego w celu przejścia ostatniego testu jako ostatecznej wersji rozwiązania. Napisz tyle kodu ile potrzeba, by test zakończył się powodzeniem!

Głównym celem stosowania TDD jest uzyskanie projektu kodu umożliwiającego przeprowadzeniem testów. Testy są bardzo przydatnym efektem ubocznym całego procesu!

Testy mogą służyć jako wykonywalna dokumnetacja, a TDD to najczęściej stosowany sposób tworzenia i konserwacji takich testów.

Jeżeli któryś z wcześniejszych testów zakończył się porażką, oznacza to, że ostatnie zmiany uszkodziły rozwiązanie. Dlatego należy wycofać modyfikację.

Pojęcie testu jednostkowego

Test jednostkowy jest fragmentem kodu sprawdzającym działanie pewnego niewielkiego, dokładnie określonego obszaru funkcjonalności testowanego kodu. Zwykle polega on na wykonaniu wybranej metody w określonym kontekście (sprawdzenie asercji).

Zadaniem testów jednostkowych jest udowodnienie, że kod działa zgodnie z założeniami programisty.

Oczywiście istnieje wiele innych rodzajów testów (funkcjonalne, integracyjne, wydajnościowe, itd.)

JUnit

JUnit to prosta w użyciu i łatwa do opanowania platforma do pisania i uruchamiania testów. Każdy test ma postać metody. Każda taka metoda powinna reprezentować określony znany scenariusz wykonywania fragmentów kodu. Sprawdzenie kodu odbywa się w wyniku porównania oczekiwanych danych wyjściowych lub działań z rzeczywistymi danymi wyjściowymi.

Oprócz **JUnit** można używać również platformę **TestNG**, z różnicą taką, że testy uporządkowane są w klasach.

Przykładowy kod

Napiszemy program, który wyświetli nam napis (np. Hello + tekst).
Najpierw piszemy test (zgodnie z zasadą TDD):

```
public class GreetingTest {
    @Test
    public void returnMessage(){
        Greeting gret = new Greeting();
        String result = gret.returnMessage("JUnit");
        assertEquals("Hello JUnit",result);
    }
}
```

Przykładowy kod

Test nie przeszedł. Następnie piszemy kod, dzięki któremu przejdzie test:

```
  public class Greeting {
    public String returnMessage(String string){
        return "Hello " + string;
      }
}
```

Teraz test przeszedł. Czy rozważyliśmy wszystkie przypadki?

Przykładu ciąg dalszy - przypadki brzegowe

Co się stanie gdy parametrem będzie słowo puste lub null? Wyrzucimy wyjątek IllegalArgumentException.

```
@Test
  public void returnNullMessage(){
      Greeting gret = new Greeting();
      String result = gret.returnMessage(null);
      assertEquals(null,result);
  }
```

```
public String returnMessage(String string){
      if (string == null){
          throw new IllegalArgumentException();
      }
      return "Hello " + string;
  }
```

Przykład - obsługa wyjątków

Test nadal nie przeszedł, bo nie obsłużyliśmy wyjątków. Poprawny test wygląda następująco:

```
@Test(expected=IllegalArgumentException.class)
  public void returnNullMessage(){
      Greeting gret = new Greeting();
      String result = gret.returnMessage(null);
      assertEquals(null,result);
  }
```

Przykład - obsługa wyjątków

Test nadal nie przeszedł, bo nie obsłużyliśmy wyjątków. Poprawny test wygląda następująco:

```
@Test(expected=IllegalArgumentException.class)
  public void returnNullMessage(){
      Greeting gret = new Greeting();
      String result = gret.returnMessage(null);
      assertEquals(null,result);
  }
```

Podobny test należy przeprowadzić w przypadku kiedy mamy pusty napis.

Test - adnotacje @Before i @After

W każdym teście tworzymy obiekt **gret**. Adnotacja `@Before` pozwala nam wywołać metodę przed wywołaniem każdego testu. Analogicznie jest z adnotacją `@After`. Poprawiony kod wygląda następująco:

```
    private Greeting gret;

    @Before
    public void setUp(){
        gret = new Greeting();
    }

    \\ Testy

    @After
    public void tearDown(){
        gret = null;
    }
```

Bibliografia


A. Hund D. Thomas, *JUnit. Pragmatyczne testy jednostkowe w Javie, Wydawnictwo Helion, 2006.*


V. Farcic, A. Garcia, *TDD. Programowanie w Javie. Sterowanie testami, Wydawnictwo Helion, 2016.*